**Conference Session Scheduling Project**

Part 1

Brandon McFarland

CSE 5350

Southern Methodist University

**1) Computing Environment**

|  |  |
| --- | --- |
| Hardware Overview | Software Overview |
| * **Model Name:** MacBook Pro * **Processor Name:** Intel Core i7 (Skylake) * **Processor Speed:** 2.6 GHz * **Number of Processors:** 1 * **Total Number of Cores:** 4 * **L2 Cache (per Core**): 256 KB * **L3 Cache:** 6 MB * **Memory:** 16 GB 2133 MHz LPDDR3 | * **Operating System:** macOS Mojave Version 10.14.3 * **Kernel:** Darwin Kernel Version 18.2.0 * **Programming Language:** Python |

**2) “Custom” Distribution**

For the distribution of my choosing I chose to implement a Gaussian (normal distribution). This distribution is created using the same built-in random number generator, numpy.random.randint(), as the other implemented distributions and is discussed at greater length in the following section (3).

**2) Methods of Distribution Creation**

Uniform Distribution

Overview:

This distribution is created simply by calling the built-in uniform distribution pseudo-random number generator passing in the value 1 as the argument for the low parameter and the number of sessions plus 1 as the argument for the high parameter as the interval for this function is half-open and exclusive on the high end of the range.

Code

|  |
| --- |
| numpy.random.randint(1, sessions + 1) |

Tiered Distribution

This distribution is created through the concatenation of separately created uniform tiers. The first tier is created by calling the built-in uniform distribution pseudo-random number generator passing in the value 1 as the argument for the low parameter, the number of sessions multiplied by 0.1 as the argument for the high parameter, and the number of sessions multiplied by 0.5 as the argument for the size parameter. The second tier is created by calling the built-in uniform distribution pseudo-random number generator passing in the number sessions multiplied by 0.1 as the argument for the low parameter, the number of sessions plus 1 as the argument for the high parameter, and the number of sessions multiplied by 0.5 as the argument for the size parameter. The resulting arrays from the creation of the first and second tiers are then concatenated, thus the first 10% of the sessions have 50% of the probability distribution and the remaining 90% of the sessions have 50% of the probability distribution.

Code

|  |
| --- |
| first = np.random.randint(1, int(sessions \* 0.1), int(sessions \* 0.5))  second = np.random.randint(int(sessions \* 0.1) , sessions + 1, int(sessions \* 0.5))  final = np.append(first, second) |

Skew Distribution

Overview:

This distribution is created through mathematical manipulation of two uniform distributions. In order to create a skewed distribution, I called the built-in uniform distribution pseudo-random number generator twice passing in the value 1 as the argument for the low parameter, the number of sessions plus 1 as the argument for the high parameter both times. I then took the absolute value of the difference of the two distribution arrays, resulting in a skewed distribution.

Code

|  |
| --- |
| first = np.random.randint(1, sessions + 1)  second = np.random.randint(1, sessions + 1)  final = abs(a-b) |

Custom Distribution (Gaussian)

Overview:

This distribution is also created through mathematical manipulation of a uniform distribution. First I set the mean of the distribution to be the number of session divided by 2 and set sigma to be the mean multiplied by 0.4. I then created a uniform distribution by calling the built-in uniform distribution pseudo-random number generator passing in the value 1 as the argument for the low parameter, the number of sessions plus 1 as the argument for the high parameter. After creating my distribution I loop over the values contained therein and divide each value by the number of sessions + 1, resulting in floating point values between 0 and 1. The Gaussian distribution is then created by plugging the created values into the following formula:



Where is the inverse error function:



Code

|  |
| --- |
| mean = sessions / 2  sigma = mean \* 0.4  uniform = np.random.randint(1,sessions + 1)  x = [ float(i) / sessions for i in uniform]  final = mean + 2\*\*0.5 \* sigma \* erfinv(np.array(x) \* 2 - 1) |

Histograms

|  |  |
| --- | --- |
|  |  |
|  |  |

**4) Space and Time Complexity of Distribution Creation**

Theory

|  |  |  |
| --- | --- | --- |
| With respect to N | Space Complexity | Time Complexity |
| Uniform |  |  |
| Tiered |  |  |
| Skew |  |  |
| Custom (Normal) |  |  |

|  |  |  |
| --- | --- | --- |
| With respect to S | Space Complexity | Time Complexity |
| Uniform |  |  |
| Tiered |  |  |
| Skew |  |  |
| Custom (Normal) |  |  |

|  |  |  |
| --- | --- | --- |
| With respect to K | Space Complexity | Time Complexity |
| Uniform |  |  |
| Tiered |  |  |
| Skew |  |  |
| Custom (Normal) |  |  |

Data

|  |  |
| --- | --- |
|  |  |
|  |  |

**5) Duplicate Removal Algorithms**

Algorithm 1

The first algorithm I implemented for removing duplicate conflicts makes use of the Python set object, an unordered collection of distinct hashable values. The strategy for this algorithm was to create a set of sets resulting in a distinct group of pairwise conflicts. Conflicts are passed to the duplicate removal function as a list of lists where each sublist represents a distinct pairwise conflict. In order to represent sets of sets, the inner sets must be frozenset object so this is where I began. I use a list comprehension to iterate over the sublists, converting each sublist to a frozenset, I then call set() on the list of frozenset objects, resulting in a distinct group of pairwise conflicts. This algorithm has space and time complexity of where n is the number of sublists as operations are done in place.

Algorithm 2

The second algorithm I implemented for removing duplicate conflicts makes use of a Python dictionary to “remember” which conflicts have already been seen. Conflicts are passed to the duplicate removal function as a list of lists where each sublist represents a distinct pairwise conflict. The following actions occur on each iteration as pairs of sublists are evaluated: before evaluating the conflict pair the elements of the pair are tested and manipulated to ensure the smaller value in the pair is element0. If neither element is a key of the distinctConflicts dictionary add key element0 to distinctConflicts and set the value to an empty list then append the value of element1 to the element0 list. Else, if one of the elements is already a key in distinctConflicts append the value of the other element to the existing keys list. Otherwise the current pair is a duplicate and the algorithm does nothing. This algorithm has space complexity of . And time complexity of where n is the number of sublists as it involves doing linear searches in the target lists to check for membership.

**6) Space and Time Complexity of Duplicate Removal**

Theory

|  |  |  |
| --- | --- | --- |
| With respect to N | Space Complexity | Time Complexity |
| Algorithm 1 |  |  |
| Algorithm 2 |  |  |

|  |  |  |
| --- | --- | --- |
| With respect to S | Space Complexity | Time Complexity |
| Algorithm 1 |  |  |
| Algorithm 2 |  |  |

|  |  |  |
| --- | --- | --- |
| With respect to K | Space Complexity | Time Complexity |
| Algorithm 1 |  |  |
| Algorithm 2 |  |  |

|  |  |  |
| --- | --- | --- |
| With respect to M | Space Complexity | Time Complexity |
| Algorithm 1 |  |  |
| Algorithm 2 |  |  |

Data

**7) Program Output Examples**

**8) Appendix**

Source code for part 1

|  |
| --- |
| **import math**  **import time**  **import matplotlib**  **import numpy as np**  **matplotlib.use(*'TkAgg'*)**  **from pdb import set\_trace as bp**  **from scipy.special import erfinv**  **from collections import defaultdict**  **from memory\_profiler import profile**  **from matplotlib import pyplot as plt**  **from timeit import default\_timer as timer**  ***# generate and return e and p array***  **def generateEPArrays(conflictAdjacency):**  ***# init arrays with single element, 0, to begin with session number 1***  **e = []**  **p = []**  **eIndex = 0**  **pIndex = 0**  **for key, value in conflictAdjacency.iteritems():**  **if len(value) == 0:**  **continue**  **else:**  **p.append(eIndex)**  **for i in value:**  **e.append(i)**  **eIndex += 1**  **return [e, p]**  ***# generate an adjacency list for creation of E[] and P[]***  **def generateAdjacency(sessions, distinctConflicts):**  **conflictAdjacency = {}**  **conflictKeys = distinctConflicts.keys()**  ***# initalize lists for all sessions***  **for i in range(1, sessions + 1):**  **conflictAdjacency[i] = []**  **for key in conflictKeys:**  **currentConflicts = distinctConflicts[key]**  **for session in currentConflicts:**  **conflictAdjacency[key].append(session)**  **if key not in conflictAdjacency[session]:**  **conflictAdjacency[session].append(key)**  **return conflictAdjacency**  **def calculateDistinctConflicts2(conflicts):**  **distinctConflicts = set(frozenset(pair) for pair in conflicts)**  **return distinctConflicts**  **def calculateDistinctConflicts1(conflicts):**    ***# create a dictionary for distinct conflicts***  **distinctConflicts = {}**  **for pair in conflicts:**  ***# make the smallest value element 0***  **if pair[0] > pair[1]:**  **element0 = pair[0]**  **element1 = pair[1]**  **else:**  **element0 = pair[1]**  **element1 = pair[0]**  ***# create a list of the keys that are in the dictionary***  **conflictKeys = distinctConflicts.keys()**  ***# if neither element is a key of the distinctConflicts dictionary***  **if element0 not in conflictKeys and element1 not in conflictKeys:**  ***# add key element0 to distinctConflicts and set the value to an empty list***  **distinctConflicts[element0] = []**  ***# assign a var to the new empty list***  **targetList = distinctConflicts[element0]**  ***# append the value of element1 to element0's list***  **targetList.append(element1)**  ***# if one of the elements is already a key in distinctConflicts append the value of the other***  ***# element to the existing keys list***  **elif element0 in conflictKeys and element1 not in distinctConflicts[element0]:**  **targetList = distinctConflicts[element0]**  **targetList.append(element1)**  **elif element1 in conflictKeys and element0 not in distinctConflicts[element1]:**  **targetList = distinctConflicts[element1]**  **targetList.append(element0)**  ***# Otherwise this is a dupicate, do nothing***  **elif element0 in conflictKeys and element1 in distinctConflicts[element0] or element1 in conflictKeys and element0 in distinctConflicts[element1]:**  ***# duplicate, don't add***  **continue**  **return distinctConflicts**  **def calculateConflicts(schedule):**  ***# create a list to hold the lists of pairwise conflicts***  **conflictArray = []**  **for key, value in schedule.iteritems():**  ***# grab the session list of the attendee***  **currentList = schedule[key]**  ***# create lists of pairwise conflicts between attendee sessions***  **for i in range(0, len(currentList) - 1):**  **for x in range(i, len(currentList) - 1):**  ***# add the pairwise conflict list to the conflict array***  **conflictArray.append([currentList[i], currentList[x+1]])**  **return conflictArray**  **def createUniform(sessions, attendees):**  **dist = np.random.randint(1, sessions + 1, *size*=sessions\*attendees)**  **dist = dist.tolist()**  ***# code for plotting the distribution***  ***# data = np.random.randint(1, 101, 2500)***  ***# count, bins, ignored = plt.hist(data , 50, facecolor='green')***  ***# plt.xlabel('X~U[1,100]')***  ***# plt.ylabel('Count')***  ***# plt.title("Uniform Distribution Histogram (Bin size 2)")***  ***# plt.axis([1, 100, 0, 100])***  ***# plt.grid(True)***  ***# plt.show(block=True)***  **return dist**  **def createTiered(sessions):**  **first = np.random.randint(1, int(sessions \* 0.1), int(sessions \* 0.5))**  **second = np.random.randint(int(sessions \* 0.1) , sessions + 1, int(sessions \* 0.5))**  **dist = np.append(first, second)**  **dist = dist.tolist()**  ***# code for plotting the distribution***  ***# data = np.random.choice(final, sessions)***  ***# count, bins, ignored = plt.hist(dist, int(sessions//10), facecolor='green')***  ***# plt.xlabel('X~U[1,500]')***  ***# plt.ylabel('Count')***  ***# plt.title("Tiered Distribution Histogram (Bin size 50)")***  ***# plt.axis([1, sessions, 0, 70])***  ***# plt.grid(True)***  ***# plt.show(block=True)***  **return dist**  **def createSkewed(sessions):**  **a = np.random.randint(1, sessions + 1, sessions)**  **b = np.random.randint(1, sessions + 1, sessions)**  **dist = abs(a-b)**  **dist = dist.tolist()**  ***# code for plotting the distribution***  ***# a = np.random.randint(1, 101, 2500)***  ***# b = np.random.randint(1, 101, 2500)***  ***# c = abs(a-b)***  ***# count, bins, ignored = plt.hist(c, 50, facecolor='green')***  ***# plt.xlabel('X~U[1,100]')***  ***# plt.ylabel('Count')***  ***# plt.title("Skewed Distribution Histogram (Bin size 2)")***  ***# plt.axis([1, 100, 0, 100])***  ***# plt.grid(True)***  ***# plt.show(block=True)***  **return dist**  **def createCustom(sessions):**  **mean = sessions / 2**  **sigma = mean \* 0.4**  **uniform = np.random.randint(1,sessions + 1, sessions)**  **x = [ float(i) / (sessions+1) for i in uniform]**  **dist = mean + 2\*\*0.5 \* sigma \* erfinv(np.array(x) \* 2 - 1)**  **dist = dist.tolist()**  ***# code for plotting the distrib0ution***  ***# mean = 50***  ***# sigma = 20***  ***# a = np.random.randint(1,100,1500)***  ***# x = [ float(i) / 100 for i in a]***  ***# x = mean + 2\*\*0.5 \* sigma \* erfinv(np.array(x) \* 2 - 1)***  ***# count, bins, ignored = plt.hist(dist, 50, facecolor='green')***  ***# plt.xlabel('X~U[1,100]')***  ***# plt.ylabel('Count')***  ***# plt.title("Custom Distribution Histogram (Bin size 2)")***  ***# plt.axis([1, 100, 0, 100])***  ***# plt.grid(True)***  ***# plt.show(block=True)***  **return dist**  **def generateSchedule(sessions, attendees, maxSessions, distribution, attendeeDict):**  **print *'Generating schedule...'***  ***# uniform case***  **if distribution == *'uniform'*:**  ***# I referenced the following url for uniform distribution code***  ***# https://stackoverflow.com/questions/22744577/plotting-basic-uniform-distribution-on-python***  **print *'Creating the uniform distribution...'***  **uniformDistribution = createUniform(sessions, attendees)**  **print *'Sampling from uniform distribution...'***  ***# loop over the keys in the dict***  **for key, value in attendeeDict.iteritems():**  ***# start with index 0***  **currentIndex = 0**  ***# for each session space for the attendee...***  **for s in attendeeDict[key]:**  ***# if there is a default space in the array...***  **if None in attendeeDict[key]:**  ***# select a session using the uniform distribution***  **selection = np.random.choice(uniformDistribution, 1)**  **selection = selection[0]**  ***# while the selected session in already in the list...***  **while selection in attendeeDict[key]:**  ***# make a new selection***  **selection = np.random.choice(uniformDistribution, 1)**  **selection = selection[0]**  ***# break when the selected session is not already in the list***  **if selection not in attendeeDict[key]:**  **break**  ***# add the session to the attendees list***  **attendeeDict[key][currentIndex] = selection**  ***# move to the next session space***  **currentIndex += 1**  **else:**  **print *'ERROR - Uniform dist index error'***  **exit()**  **return attendeeDict**  **elif distribution == *'tiered'*:**  **print *'Creating the tiered distribution...'***  **tieredDistribution = createTiered(sessions)**  **print *'Sampling from tiered distribution...'***  ***# loop over the keys in the dict***  **for key, value in attendeeDict.iteritems():**  ***# start with index 0***  **currentIndex = 0**  ***# for each session space for the attendee...***  **for s in attendeeDict[key]:**  ***# if there is a default space in the array...***  **if None in attendeeDict[key]:**  ***# select a session using the tiered distribution***  **selection = np.random.choice(tieredDistribution, 1)**  **selection = selection[0]**  ***# while the selected session in already in the list...***  **while selection in attendeeDict[key] or selection == 0:**  ***# make a new selection.***  **selection = np.random.choice(tieredDistribution, 1)**  **selection = selection[0]**  ***# break when the selected session is not already in the list***  ***# NOTE: This while loop takes a LONG TIME when sessions == maxSessions***  **if selection not in attendeeDict[key] and selection != 0:**  **break**  ***# add the session to the attendees list***  **attendeeDict[key][currentIndex] = selection**  ***# move to the next session space***  **currentIndex += 1**  **else:**  **print *'ERROR - Tiered dist index error'***  **exit()**  **return attendeeDict**  **elif distribution == *'skewed'*:**  ***# I reference the following url for skewed distribution code***  ***# https://gamedev.stackexchange.com/questions/116832/random-number-in-a-range-biased-toward-the-low-end-of-the-range***  **print *'Creating the skewed distribution...'***  **skewedDistribution = createSkewed(sessions)**  **print *'Sampling from skewed distribution...'***  ***# loop over the keys in the dict***  **for key, value in attendeeDict.iteritems():**  ***# start with index 0***  **currentIndex = 0**  ***# for each session space for the attendee...***  **for s in attendeeDict[key]:**  ***# if there is a default space in the array...***  **if None in attendeeDict[key]:**  ***# select a session using the skewed distribution***  **selection = np.random.choice(skewedDistribution, 1)**  **selection = selection[0]**  ***# while the selected session in already in the list...***  **while selection in attendeeDict[key] or selection == 0:**  ***# make a new selection.***  **selection = np.random.choice(skewedDistribution, 1)**  **selection = selection[0]**  ***# break when the selected session is not already in the list***  ***# NOTE: This while loop takes a LONG TIME when sessions == maxSessions***  **if selection not in attendeeDict[key] and selection != 0:**  **break**  ***# add the session to the attendees list***  **attendeeDict[key][currentIndex] = selection**  ***# move to the next session space***  **currentIndex += 1**  **else:**  **print *'ERROR - Skewed dist index error'***  **exit()**  **return attendeeDict**  **elif distribution == *'custom'*:**  ***# I referenced the following url for normal distribution code***  ***# https://stackoverflow.com/questions/38754423/drawing-gaussian-random-variables-using-scipy-erfinv***  **print *'Creating the custom distribution...'***  **customDistribution = createCustom(sessions)**  **print *'Sampling from custom distribution...'***  ***#loop over the keys in the dict***  **for key, value in attendeeDict.iteritems():**  ***# start with index 0***  **currentIndex = 0**  ***# for each session space for the attendee...***  **for s in attendeeDict[key]:**  ***# if there is a default space in the array...***  **if None in attendeeDict[key]:**  ***# select a session using the custom distribution***  **selection = np.random.choice(customDistribution, 1)**  **selection = selection[0]**  ***# while the selected session in already in the list...***  **while selection in attendeeDict[key] or selection == 0:**  ***# make a new selection.***  **selection = np.random.choice(customDistribution, 1)**  **selection = selection[0]**  ***# break when the selected session is not already in the list***  **if selection not in attendeeDict[key] and selection != 0:**  **break**  ***# add the session to the attendees list***  **attendeeDict[key][currentIndex] = selection**  ***# move to the next session space***  **currentIndex += 1**  **else:**  **print *'ERROR - Normal dist index error'***  **exit()**  ***# print 'Done generating schedule'***  **return attendeeDict**  **def main():**  ***# get input values from the command line***  **sessions = raw\_input(*'Enter the number of sessions (integer): '*)**  **attendees = raw\_input(*'Enter the number of attendees (integer): '*)**  **maxSessions = raw\_input(*'Enter the number of sessions per attendee (integer): '*)**  **distribution = raw\_input(*'Choose a distribution: uniform | tiered | skewed | custom: '*)**  **conflictMethod = raw\_input(*'Choose a method to remove conflicts: 1 | 2: '*)**  ***# validate command line input***  **try:**  **sessions = int(sessions)**  **except ValueError as e:**  **print *'ERROR - Number of sessions must be an integer'***  **exit()**    **try:**  **attendees = int(attendees)**  **except ValueError as e:**  **print *'ERROR - Number of attendees must be an integer'***  **exit()**  **try:**  **maxSessions = int(maxSessions)**  **except ValueError as e:**  **print *'ERROR - Max number of sessions must be an integer'***  **exit()**    **try:**  **distribution = distribution.lower().replace(*' '*, *''*)**  **except error as e:**  **print *'ERROR - Unknown error in distribution format'***  **exit()**  **if distribution != *'uniform'* and distribution != *'tiered'* and distribution != *'skewed'* and distribution != *'custom'*:**  **print *'ERROR - Invalid distribution type'***  **exit()**  **if conflictMethod != *'1'* and conflictMethod != *'2'*:**  **print *'ERROR - invalid input for conflict method'***    ***# adjust max sessions for skewed an custom distributions***  **if distribution == *'skewed'* or distribution == *'custom'*:**  **maxSessions = int(0.1 \* maxSessions)**  **if maxSessions < 1:**  **maxSessions = 1**  ***# create dictionary of lists to hold the sessions for each attendee***  ***# each session spot is initialized to None***  **attendeeDict = {}**  **for i in range (0, attendees):**  **attendeeDict[i + 1] = [None] \* maxSessions**  ***# pass validated inputs to schedule generation function generateSchedule()***  ***# to get a dictionary of lists representing each attendee's schedule***  **schedule = generateSchedule(sessions, attendees, maxSessions, distribution, attendeeDict)**  ***# enumerate all conflicts***  **conflicts = calculateConflicts(schedule)**  ***# remove duplicate conflicts***  **if conflictMethod == *'1'*:**  **distinctConflicts = calculateDistinctConflicts1(conflicts)**  **elif conflictMethod == *'2'*:**  **distinctConflicts = calculateDistinctConflicts2(conflicts)**  ***# generate adjacency list***  **conflictAdjacency = generateAdjacency(sessions,distinctConflicts)**  ***# generate E and P array***  **EParrays = generateEPArrays(conflictAdjacency)**  ***# print the result!***  **print { *"sessions(N)"*: sessions,**  ***"distinctConflicts(M)"*: distinctConflicts,**  ***"conflicts(T)"*: conflicts,**  ***"attendees(S)"*: attendees,**  ***"maxSessions"*: maxSessions,**  ***"distribution"*: distribution,**  ***"eArray"*: EParrays[0],**  ***"pArray"*: EParrays[1]**  **}**  ***# test if script is being run directly***  **if \_\_name\_\_ == *'\_\_main\_\_'*:**  **main()** |